COMP251_Assignment 1_Conclusion

260713093 Agnes Liu

PART 1

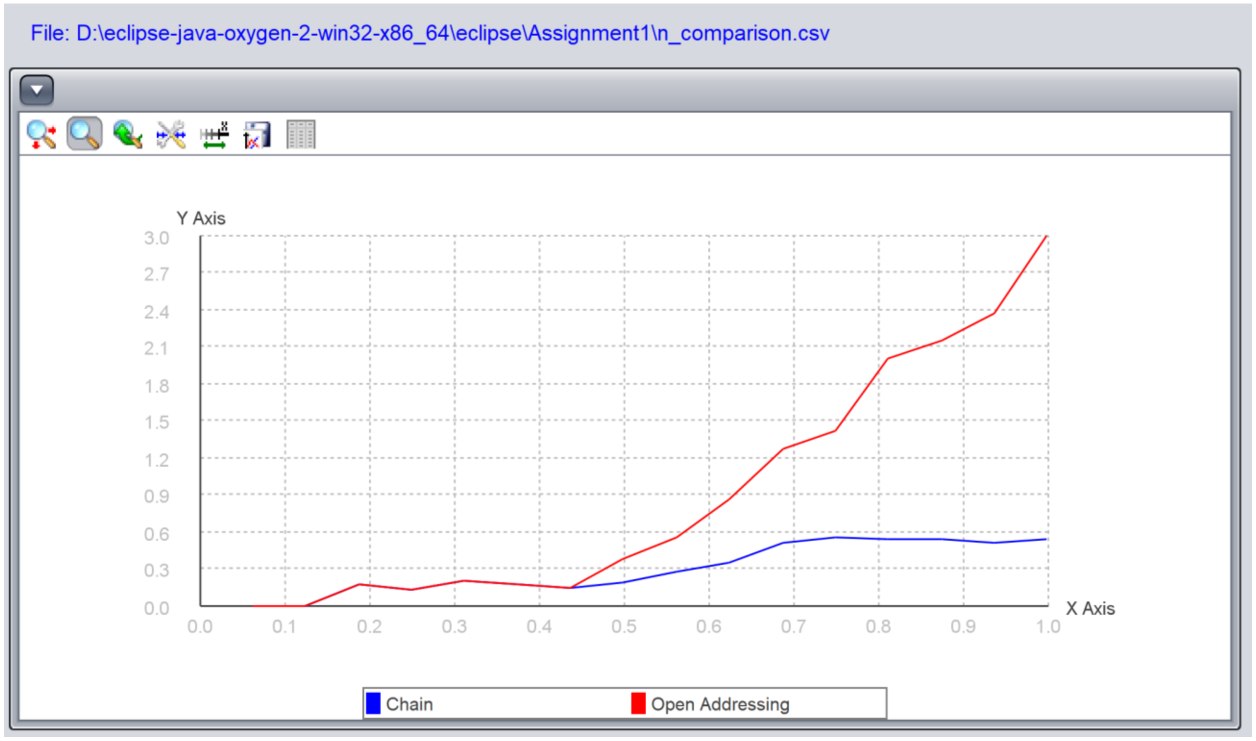


Figure 1: n_conparison.csv generated by <task 1> in <main>method
Notations: n = number of keys;
m=number of slots;
avC = average number of collisions encountered in chaining
avP = average number of collisions encountered in open-addressing

Figure 1 shows how avC and avP change respectively as n inserted increases. From the graph it is obvious that both avC and avP grow as n increases. Also, as n becomes larger and larger, the value for avP-avC becomes more and more significant, meaning open-addressing would have more collisions than chaining in a larger number of key-insertion.

In chaining, we implemented a multidimensional arraylist to represent the hash table, in which if an index (0<=index<=m-1) of the table matches with the chaining hash function value for a certain key, we simply append the key to the arraylist indicated by that index. The number of elements in such arraylist before the insertion is the number of collisions for this chaining insertion.

In open-addressing, we implemented an array with m slots to represent the hash table, in which insertion will happen in one of these 2 choices:

1. The index with same value as the key's hash function value indicates an empty slot→ insert the key into the slot;

2. The index with same value as the hash function is full → insert the key into the first empty slot along the probing.

From the logics of the 2 methods we can conclude that inserting a key by chaining is only considering other keys with the same hash function as its collisions, while inserting by open-addressing is considering 1 key with the same hash function + all other keys encountered during linear probing as its collisions. As n increases, the likelihood for repetitive hash function values is also increasing, forcing open-addressing to check the emptiness of significantly more slots than chaining.

PART2

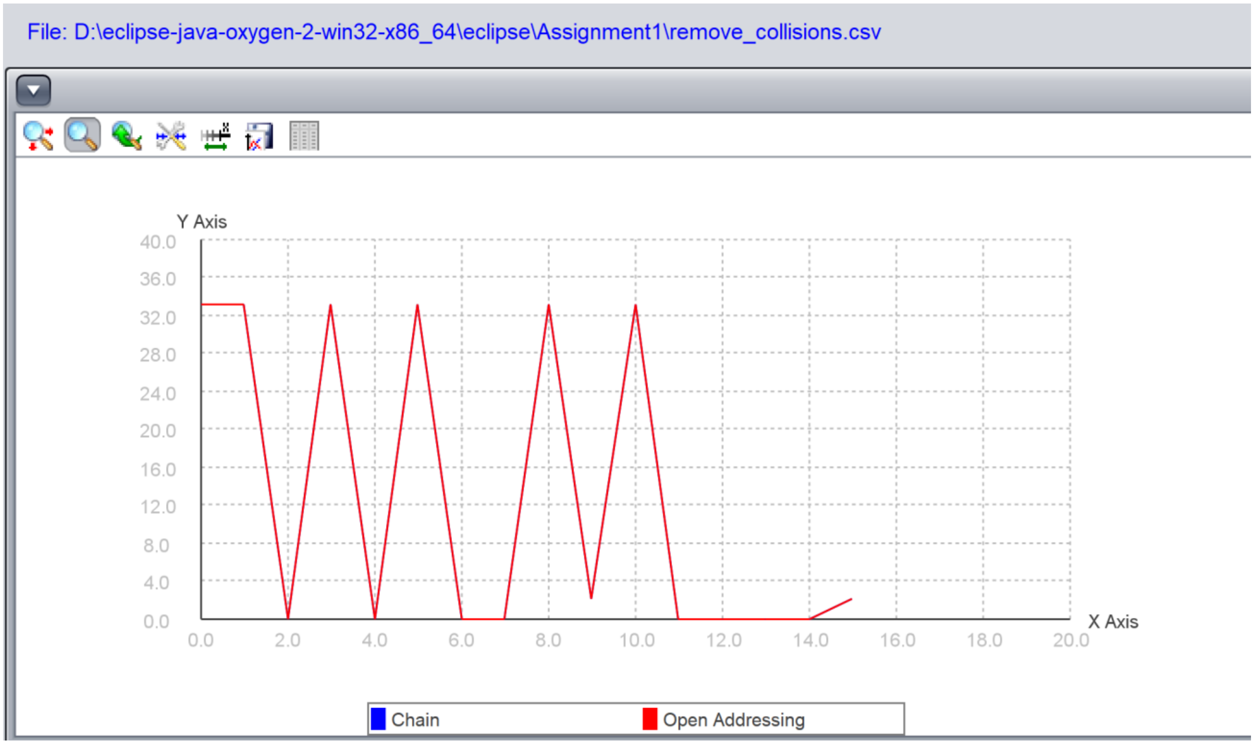


Figure 2: remove_collisions.csv generated by <task 2> in <main>method

Index	0	1	2	3	4	5	6	7	8
Key	6	8	164	180	127	3	481	132	4
collisions	33	33	0	33	0	33	0	0	33

Index	9	10	11	12	13	14	15
Key	467	5	160	205	186	107	179
collisions	2	33	0	0	0	0	2

Charts 1&2: key indexes (x-axis in figure2), keys, and collisions(y-axis in figure2)

From task 2 of the main method, my newOpenAddressing.Table looks like this after all 16 insertions:

[533, 207, 810, -1, -1, -1, -1, 164, -1, -1, 858, 132, 160, -1, -1, 107, -1, 205, -1, -1, 906, 955, -1, -1, -1, 127, 481, 467, -1, -1, 186, 179]

Clearly none of the keys with collision =33 was inserted into the array, which is why the method returned 33 (the total number of slots);

Everything with collisions=0 are keys inserted right at the slot with index==h(key), and collisions of 2 meaning given the corresponding hashing slot full, the method searched for 1 other full slot before it found the key along the open-addressing logic.

<u>PART 3</u>

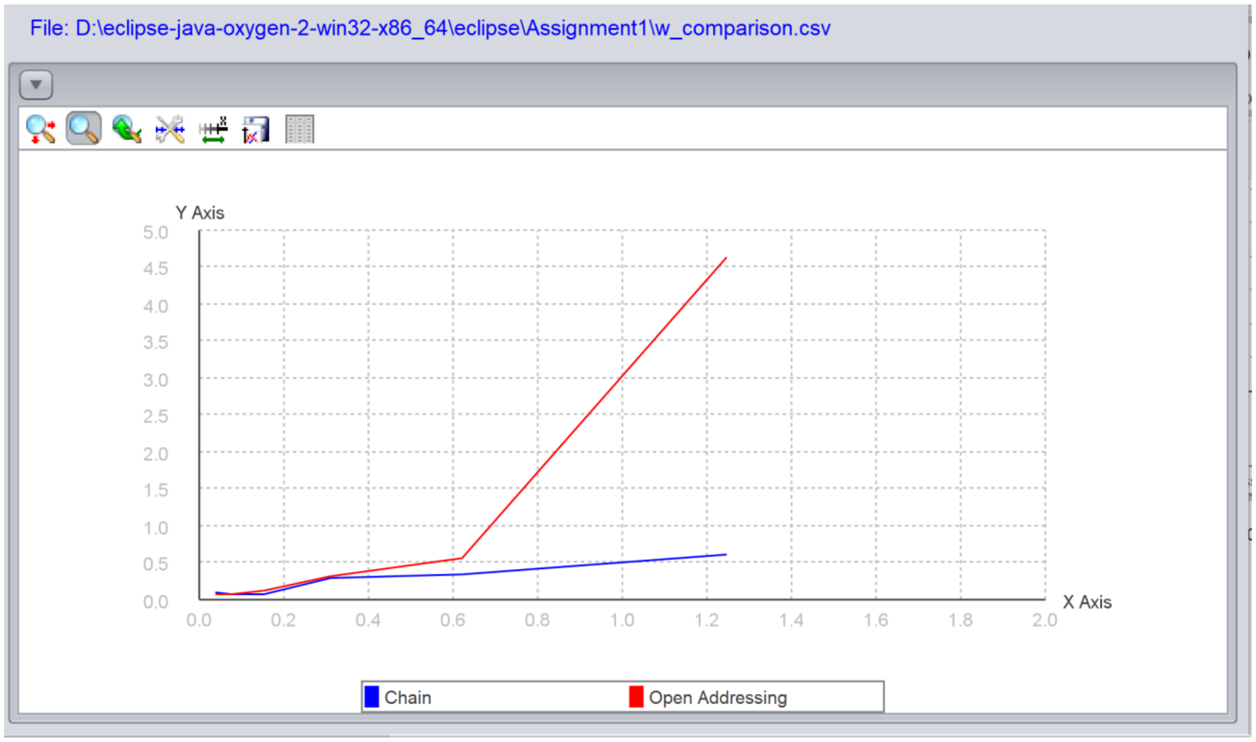
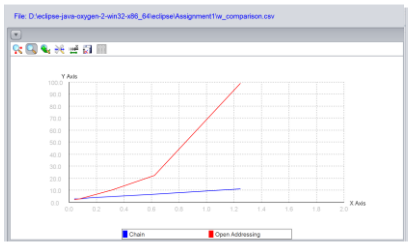
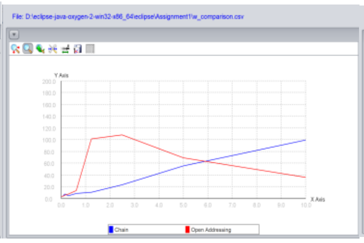


Figure 3: w_comparison.csv generated by <task 3> in <main>method w list: {8,10,12,14,16,18} average divided by (10*n)

This is the main graph for my task 3 (graph generated by the current settings). With collisions averages calculated by dividing (10*n), the graphs generated were more static comparing to ones divided by 10 only. From Figure 3 and all graphs of other choices listed below, numbers of collisions **experience general growth as w increases. For chaining, the increase of collisions is in linear behavior, for the value of w does not influence collisions it encounters as significant as number of keys with the same hash values. For Open addressing, the increase is often more dramatic due to a similar reason of part 1.** However, in graphs with the widest range of w, the increasing behavior on collision number for open addressing started to drop. This may because of the fact that significantly more slots than keys would reduce the number of collisions open addressing run into.





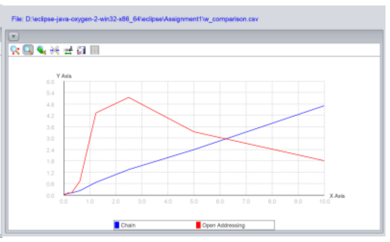
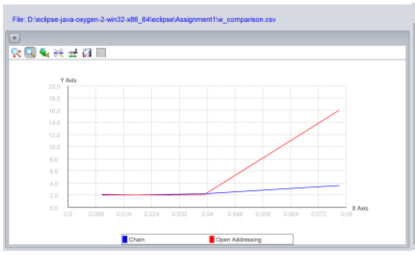
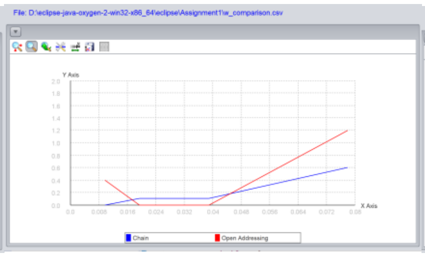


Figure 4,5,6 (left to right) wlist: {8,10,12,14,16,18} divided by 10 only (figure4) and
w list{2,4,6,8,10,12,14,16,18,20,22,24} divided by 10 only (figure 5) and by(10*n) (figure 6)





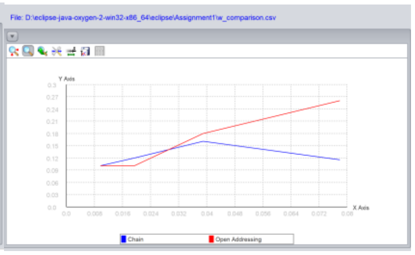


Figure 7,8,9 (left to right). W list: {16,18,20,22} divided by 10 only (figure 7,8) and divided by (10*n)